DISTLIB
A Library for Message-Based Distributed Programs

Dennis Heimbigner

CU-CS-352-86 March 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency and the IBM Corporation.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **MAR 1988**	2. REPORT TYPE			3. DATES COVERED **00-00-1988 to 00-00-1988**	
4. TITLE AND SUBTITLE **DISTLIB: A Library for Message-Based Distributed Programs**				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado,Department of Computer Science,Boulder,CO,80309**				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR’S ACRONYM(S)	
				11. SPONSOR/MONITOR’S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**21**	

1. Introduction and Philosophy

Distlib is a set of library routines that provides a framework for message based distributed programs. It uses the stream socket facilities of Berkeley Unix 4.2 as its basis, but provides a set of additional features to make distributed programming simpler.

This library assumes that the distributed program is actually made up of a collection of process types instantiated on some subset of the nodes of a network. It explicitly rejects the notion of anonymous process. Processes communicate with each other in terms of (Node,Process-type) addresses. All communications between processes is assumed to be in the form of messages. Messages may be variable length up to some fixed bound. User processes are event driven and run-to-completion, which means that when a message arrives, it is read initially by the distributed library event handle and then passed to a user-defined procedure to be processed. When the user procedure returns, the event handler waits to receive another message from any source. This cycle of receive-process is repeated indefinitely.

Inter-process communication is connection-based. This means that when one process on one node, say (N1,P1) wishes to communicate with another process on another (or same) node, say, (N2,P2), it must open a connection to that process on that node and send messages over that connection. This results in the creation of a connection identifier[1] (an integer) that may be used as a handle to send to that process on that node. It is also possible to use the full handle consisting of the (node,process) address pair.

Connections are assumed to be bi-directional (i.e., readable and writable by the two connected processes). However, the input end for each process is assumed to be under the control of the event handler in the distributed library. User procedures are free to write on the output end on each process using the message sending facilities of the distlib

[1] For stream sockets, a connection and a stream file descriptor are equivalent. Given reliable datagrams, this equivalence can be broken, and a connection is just an internal handle for a particular (node,process) pair.

library.

2. Architecture

The distributed library imposes a particular structure on programs that use it. It provides a main program and select-loop code to control event handling.

Figure 1 shows the components of a distributed program. The arrow show the flow of data in the program. The input lines and the output lines are intended to correspond but representing the two directions of I/O. Messages come in on the input lines and are examined by the event handler. It then passes them to user defined code for processing.

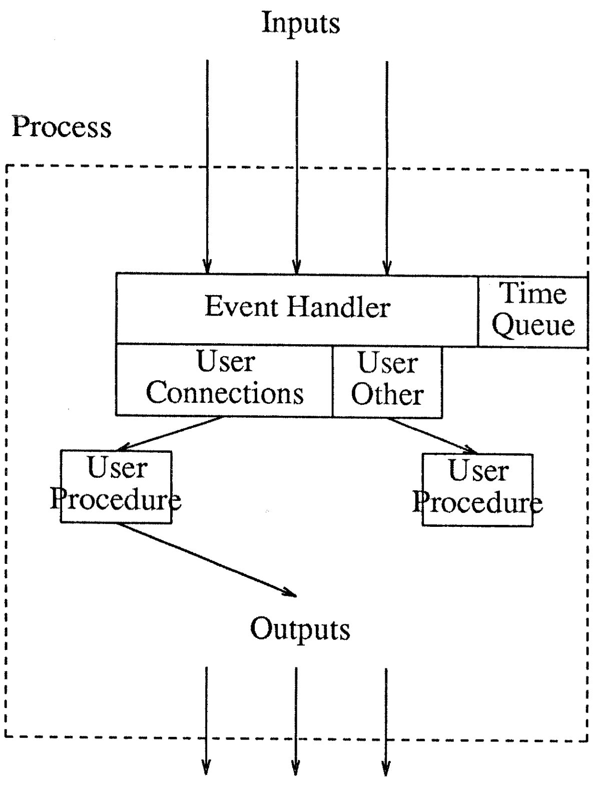

Figure 1. Distlib Process Structure.

It is normally expected that this user processing code will examine the message type and pass it to a particular procedure to handle. When finished, the procedure will return to the user handler and that in turn will return to the main event handler to await new messages.

Messages may sent over the output connections as part of the processing performed by the user procedures. Sending over a connection is assumed to have no effect on any message to read over that connection.

In addition to connections, the event handler also fields two other kinds of events: other file descriptors, and the clock. The user may register arbitrary file descriptors with the event handler. When data is available on that descriptor, a user-defined procedure is invoked to notify the user and allow the user program to handle the descriptor. this might be used, for example, to respond to pseudo-teletypes, or to commands typed in on a controlling terminal.

3. Nodes

The node database contains information about the various nodes in the network. A node is referenced by a unique identifier (type *node_t* in figure 2). A collection of information (type *node_data_t* in figure 2), is associated with each node. This structure records the name of the node, the type of machine (type *machine_t* in figure 2), and a single word of user-defined data.

Finally, distlib provides a pointer *(this_node)* to the node name for the node upon which a process is executing.

When a process starts execution, it is expected to provide in the context record (see below) an initial array of nodes to store in the node database. Two special node identifiers are predefined: UNDEFINED_NODE and ALL_NODE. They have no built-in semantics, but may be convenient as arguments to or results from functions.

```
#defineNODE_NAME_LEN  255
typedef char   NODE_NAME[NODE_NAME_LEN+1];

typedef int node_t;
#define UNDEFINED_NODE        0
#define ALL_NODE  -1

typedef int machine_t
#define MACHINE_UNDEFINED  0
#define MACHINE_VAX            1
#define MACHINE_SUN            2
#define MACHINE_PYRAMID                3

typedef struct _node_data {
        node_name nodename;
        machine_t machine;
        universal udata;
        } node_data_t,*node_data_p;

extern node_t this_node;
```

Figure 2. Node Data Structures.

```
extern node_t insert_node(entry);
        node_data_p entry;

extern int delete_node(node);
        node_t node

extern node_t gen_first_node();

extern node_t gen_next_node(lastnode);
        node_t lastnode;

extern  node_data_p get_node_data(node);
        node_t node;

extern int set_node_data(node,data);
        node_t node; node_data_p data;

extern char *nodename(node);
        node_t node;

extern node_t get_node_by_name(name);
        node_name name;
```

Figure 3. Node Access Functions.

The node database has a number of access functions (figure 3). There are functions provided for inserting and deleting nodes, stepping through the list of known nodes, accessing and modifying node attributes. The functions and their semantics are as follows:

insert_node

> Insert a new node entry into the database of known nodes. Return -1 if the node already exists, otherwise insert the entry and return 0.

delete_node

> Delete a node from the database of known nodes. Return -1 if the node does not exist, otherwise delete the entry and return 0.

gen_first_node

> Return the identifier of the first "real" node in the database. If no nodes exist, return UNDEFINED_NODE.

gen_next_node

Given the identifier of a node in the database, return the "next" node in the database, if there is one, else return UNDEFINED_NODE.

get_node_data

For a named node, return a pointer to the *node_data_t* structure associated with that node. If the node does not exist, return 0.

set_node_data

Modify the data associated with a node. If the node does not exist, return -1, else return 0.

nodename

Given a node identifier, this routine returns a pointer to the name of the node. It is used mostly for providing quick access to names for debug print outs.

get_node_by_name

Given the name of a node, return its node identifier, if any. If not found, return UNDEFINED_NODE.

4. Process Types

A database of process types is also maintained (figure 4). It is analogous to the node database, and many of its functions and data structures are similar to those for the node database. It should be noted that this database stores information about process types, not individual processes. A major difference is that for processes types, the user specifies the process type identifier.

The data associated with a process type is it name, the user-defined identifier, and a piece of user-defined data. There is also a global variable *(this_process)* that records the type of process that is running.

```
#define PROCESS_NAME_LEN      255
typedef char   PROCESS_NAME[PROCESS_NAME_LEN+1];

typedef int process_t;
#defineUNDEFINED_PROCESS      0
#define ALL_PROCESS           -1
#define SYS_DEBUG        -2
#define SYS_SERVER            -3

typedef struct _process_data {
        process_t proctype;
        process_name procname;
        universal udata;
} process_data_t, *process_data_p;

extern process_t this_process;
```

Figure 3. Process Data Structures.

```
extern int insert_process(entry);
        process_data_p entry;

extern int delete_process(process);
        process_t process

extern process_t gen_first_process();

extern process_t gen_next_process(lastprocess);
        process_t lastprocess;

extern  process_data_p get_process_data(process);
        process_t process;

extern int set_process_data(process,data);
        process_t process; process_data_p data;

extern char *processname(ptype);
        process_t ptype;

extern  process_t get_process_by_name(name);
        process_name name;
```

Figure 5. Process Access Functions.

The access functions for the process database (figure 5) are entirely analogous to those for the node database.

5. Connections

A database of connections is also maintained (figure 6). It is analogous to the node database, and many of its functions and data structures are similar to those for the node database. A connection is essentially a short handle for discussing an *address*. An *address* is a (node,process-type) pair. It is represented by the type *address_t*.

The following information is associated with a connection:

address

The address of the node and process that is on the other end of the connection.

```
typedef struct _address {
        node_t node;
        process_t proctype;
} address_t,*address_p;

typedef int connection_t;
#define UNDEFINED_CONNECTION      -1

typedef int select_e;
typedef int select_set;
#define S_UNDEFINED      0
#define S_READ           1
#define S_WRITE          2
#define S_EXCEPT   4
#define S_ANY            (S_READ | S_WRITE | S_EXCEPT)

typedef int channel_e;
typedef int channel_set;
#define C_UNDEFINED      0
#define C_SOCKET   1
#define C_LISTENER       2
#define C_OTHER          4
#define C_ANY            (C_SOCKET | C_LISTENER | C_OTHER)

typedef struct {
  address_t     address;
  channel_e     ctype;
  select_set    select_modes;
  select_set    blocked;
   universal   udata;
} conn_data_t,*conn_data_p;

extern address_t this_address;
```

Figure 6. Connection Data Structures.

```
extern int insert_connection(conn_id);
        connection_t conn_id;

extern void delete_connection(cid);
        int cid;

extern connection_t gen_first_connection();

extern connection_t gen_next_connection(lastconn);
        connection_t lastconn;

extern conn_data_p get_connection_data(conn_id);
        connection_t conn_id;

extern int set_connection_data(conn_id, entry);
        connection_t conn_id;
        conn_data_p entry;

extern char *addr_to_str(addr);
        address_p addr;

extern connection_t get_connection_by_address(addr);
        address_p addr;

extern address_p conn_address(conn_id);
        connection_t conn_id;

extern universal conn_udata(conn_id);
        connection_t conn_id;

extern boolean conn_open(conn_id);
        connection_t conn_id;

extern int call_to(addr);
        address_p addr;

extern int call_to1(addr,tries,interval);
        address_p addr;
        int tries,interval;

extern void disconnect(cid);
        connection_t cid;
```

Figure 7. Connection Access Functions.

ctype

Connections may be of several varieties (see type *channel_e* in figure 6). A connection type may be undefined, it may be a socket, it may be a listener socket, or it may be something else. It is possible to form sets of types using bit or-ing.

select_modes

This field defines the way the connection will be handled in the main select-loop of the library. A connection may be marked to appear in the read mask, the write mask, or the exception mask, or any combination of those.

blocked

A connection may be temporarily blocked from appearing in the select masks by setting this field appropriately.

udata

Allow the user to associate a single piece of arbitrary data with the connection.

The functions for the connection table (figure 7) are mostly analogous to those for the node database. There are some additional functions.

conn_address

Return a pointer to the *address_t* structure associated with a specified connection.

conn_udata

Return the user data associated with a specified connection.

conn_open

Return true if the connection is open, false otherwise.

call_to

Create a connection to a specified address. Call_to1 additionally allows the specification of the number of tries and the time interval between retries. If the connection cannot be established, return -1, else return the connection identifier. If the connection is already established, do not open a new one.

disconnect

Close a connection.

6. Message Formats

Distlib provides routines to read and write messages over a connection (figure 8). Normally, distlib is the only code to use the read routines; it uses them to read and distribute the messages to the proper user handlers. A message is assumed to be a sequence of bytes of varying length (upto a specified maximum). It is normally assumed that messages have an integer as their first 4 bytes. This integer is assumed to be the type of the message. Negative and zero integers are reserved to distlib. Positive integers are

```
extern void set_msg_type(id,str);
        int id; char *str;

extern int get_msg_type(str);
        char *str;

extern sendaddr(addr,str,strlen);
        address_p addr;
        char *str;
        int strlen;

extern recvaddr(addr,str,strlen);
        address_p addr;
        char *str;
        int strlen;

extern sendconn(conn_id,str,strlen);
        connection_t conn_id;
        char *str;
        int strlen;

extern recvconn(conn_id,str,strlen);
        connection_t conn_id;
        char *str;
        int strlen;
```

Figure 8. Message Access Functions.

assumed to be user defined. The message related routines are as follows.

set_msg_type

Given a pointer to a sequence of bytes, store (in network order) a specified message identifier.

get_msg_type

Given a pointer to a message, extract the first four bytes and return them as an integer in local format.

sendaddr

Given a pointer to an address (i.e., node and process type) and given a pointer and length for a sequence of bytes, send the bytes as a message to the specified address. On success, return the number of bytes sent. If an error occurs, return -1.

recvaddr

Given a pointer to an address (i.e., node and process type) and given a pointer and length for a sequence of bytes, Read a message into the sequence of bytes from the specified address. On success, return the number of bytes read. If an error occurs, return -1.

sendconn

Given a connection identifier and given a pointer and length for a sequence of bytes, send the bytes as a message over the specified connection. On success, return the number of bytes sent. If an error occurs, return -1.

recvconn

Given a connection identifier and given a pointer and length for a sequence of bytes, Read a message into the sequence of bytes from the specified connection. On success, return the number of bytes read. If an error occurs, return -1.

7. Timer

Distlib provides a queue of time events for the convenience of the application. The application may call *timer_start* (figure 9) to indicate that the application wishes to receive an event some number of seconds in the future. The application may associate a piece of arbitrary datum with the request. This datum will be given back to the application when it is notified of the event. After an application makes an event request, if may cancel it using *timer_stop* (figure 9). The specific event to cancel is determined by matching on the value of the user-specified datum.

8. Error Reporting

```
extern void timer_start(seconds,datum);
        int seconds; universal datum;

extern void timer_stop(datum);
        universal msg;
```

Figure 9. Timer Access Functions.

```
extern errprintf(fmt,va_alist);
        char *fmt;
        va_dcl;

extern _errprintf(fmt,ap);
        char *fmt;
        va_list ap;

extern errflush();

extern char *errperror();
```

Figure 10. Error Reporting Functions.

Distlib provides support for error handling (figure 10). It provides three basic functions: errprintf, errflush, and errperror. The functions are as follows.

Errprintf

This is a printf-like function in that it takes a variable number of arguments. The first argument is the format string, and the remaining arguments are the values to be printed. Its default action is to print onto stderr.

Errflush

This functions is called to indicate the end of an error message sequence.

Errperror

Replaces *perror()* so that its output will be diverted to the error sink.

The application is free to replace these functions with others. For example, rather than sending errors to standard output, the application might wish to send them to some central logging process. Distlib itself uses these routines, so the application must be prepared to handle distlib messages. Distlib messages are preceded by a special header string so they may be detected and treated separately.

9. Debugging Support

Distlib provides a rudimentary set of debugging functions (figure 11). Applications may provide messages that are tagged with a level number to indicate severity/warning/informativeness. The following is an example of levels:

0 = no checking, also may used for true error output
1 = check process startup
2 = check gross activity flow
3 = check detailed process flow
4 = check everything

Each level includes those below it so that as the level gets higher, more and more diagnostics are printed. The global variable *dblevel* determines the level of messages to be actually printed. Messages tagged with levels above the value in *dblevel* will be

```
extern int dblevel;

extern char *dbhdr;

extern dbprintf(level,fmt,va_alist);
        int level;
        char *fmt;
        va_dcl

extern dbmore(level,fmt,va_alist);
        int level;
        char *fmt;
        va_dcl

extern dbperror(msg);
        char *msg;

extern dbfatal(fmt,va_alist);
        char *fmt;
        va_dcl
```

Figure 11. Debug Reporting Functions.

suppressed.

In addition, all debug output is preceded by the user-defined header string pointed to by the global variable *dbhdr*.

As provided, all output from these routines is passed thru the error routines, so the error routines may be redirected to redirect the debug output.

The debug routines consist of several routines.

dbprintf

Perform printf style output. The level tag is specified by the first argument. When printed, the message is tagged by the source node and process type.

dbmore

Normally, the application uses *dbprintf* to print out a single line of information. If the output is to be multi-line, then each additional line should use *dbmore* for those additional lines.

dbperror

Rather than calling perror, the application should call this function so that the output will be diverted to the same place as other debugging output.

dbfatal

If the application detects an unrecoverable error, ti can call this routine to output it and then do an orderly shutdown of the process.

10. Initialization

```
extern void distlib_init(cxt);
        context_p cxt;

extern int distlib_select();
```

Figure 12. Initialization Functions.

```
typedef struct {
        char *this_process_name;
        int msgsize;
        int process_count;
        process_data_p processes;
        int node_count;
        node_data_p nodes;
        int (*process_other)();
        int (*process_user_timeout)();
        int (*process_user_msg)();
        int (*user_listen_from)();
        } context;
```

Figure 13. Context Data Structure.

```
process_user_msg(cid,msg,msglen)
        connection_t cid; char *msg; int msglen;

process_other(descriptor,modes)
        int descriptor,modes;

process_user_timeout(datum)
        universal datum;

user_listen_from(cid)
        connection_t cid;
```

Figure 14. User Provided Functions.

To use distlib, the user's application code must perform to basic steps. First, it must invoke *distlib_init* (see figure 12) to allow distlib to initialize it databases and to establish an initial listening connection. Obviously, the application code must call this before calling any other distlib function. The *distlib_init* function takes a pointer to a context structure (see *context_t* in figure 13). The context structure allows the application to provide environmental information to distlib. This information consists of data plus user provided functions. The data items provided by a context are as follows.

This_process_name

The process type name of this process.

Msgsize

The maximum size of messages that can be received.

Process_count

A table of initial process types. Each entry in the table is of type *process_data_t.*

Node_count

A table of initial nodes. Each entry in the table is of type *node_data_t.*

In addition to the data items, the application must provide pointers to user defined functions to be called by distlib when significant events occur. The effective declarations of these functions is shown in figure 14. The semantics of these functions is as follows:

Process_user_msg

Whenever distlib detects that input is available on a socket channel, it uses *readconn()* to read a message from that channel. It examines the message type and if it is a message type defined by distlib, then it is handled internally to distlib. If it is a user-defined message type, it is passed to this routine along with the connection identifier from which the message was read.

Process_other

When input or output or exception is detected on an "other" type connection, the connection identifier and the modes (input, output, and exception) are passed to this function. Distlib does no I/O on such connections.

Process_user_timeout

The application may use the timing queue facilities (see section ?) to arrange notification at some future time. If that time event arrives, then this routine is called with the user-defined data item specified at the call to *timer_start*.

User_listen_from

When a process receives a connection request from another process, it stores the connection and processes a system message from the other process indicating the address of the calling process. In addition, it call this function to allow the user to do any application specific operations for a new connection.